Specifying and Reading Program Input with NIDR

David M. Gay
Sandia National Laboratories*, Albuquerque NM 87185, USA

March 24, 2010

Abstract

NIDR ("New Input Deck Reader") is a facility for processing input to large programs, such as DAKOTA, a program that facilitates uncertainty quantification and optimization. NIDR was written to simplify maintenance of DAKOTA, provide better checking of input, and allow use of aliases in that input. While written to support DAKOTA input conventions, NIDR could easily be used to control other programs. This paper describes NIDR and explains the algorithm NIDR uses to permit relaxed ordering of its input.

1 Introduction

DAKOTA [2, 8] is a large program with many possible behaviors, which are controlled by an input file containing various keywords and, often, associated numerical or string values. For example, Figure 1 shows input for solving the Rosenbrock test problem [14] as a least-squares problem, with computation of the least-squares residual vector done by a separate program (the "analysis_driver") called "rosenbrock". The solver (built into DAKOTA, and known in DAKOTA parlance as a "method") is `nl2sol` [4, 5], which is allowed to run for up to 50 iterations and should stop when a suitable measure of solution quality (e.g., the relative change in the sum of squares yet possible in a quadratic model of the problem) is less than convergence tolerance 10^{-4}; the solver seeks values for a vector of two "continuous design" variables whose initial value is (–1.2, 1.0) and which must both lie in the interval [–2.0, 2.0]. The residual vector (the sum of squares of whose components is to be minimized) has two

*Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin company, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE–AC04–94AL85000. This manuscript (SAND2010–1791P) has been authored by a contractor of the U.S. Government under contract DE–AC04–94AL85000.

components. The analysis driver provides the requisite first derivatives ("analytic_gradients"), but no second derivatives are to be provided ("no_hessians"). Labels "x1" and "x2" will appear in the input to the "rosenbrock" program. As illustrated in Fig. 1, DAKOTA input files can contain comments that start with "#" and extend through the rest of the line; commas or white space can separate keywords and values, and an equals sign ("=") is optional between a keyword and its associated value or values (with some keywords having no associated values, others one or several). DAKOTA recognizes several top-level keywords ("interface", "method", "model", "responses", "strategy", and "variables", not all of which need be present, and some of which can appear more than once). Other keywords, the choice of which depends on the top-level keyword, can follow a top-level keyword, and some of these keywords, in turn, recursively enable still other keywords to appear. In all, DAKOTA recognizes over 900 keywords.

```
# extracted from $DDKOTA/test/dakota_rosenbrock.in
method,
        nl2sol
        max_iterations = 50
        convergence_tolerance = 1e-4

variables,
        continuous_design = 2
        cdv_initial_point  -1.2  1.0
        cdv_lower_bounds   -2.0 -2.0
        cdv_upper_bounds    2.0  2.0
        cdv_descriptor      'x1' 'x2'

interface,
        system
        analysis_driver = 'rosenbrock'

responses,
        num_least_squares_terms = 2
        analytic_gradients
        no_hessians
```

Figure 1: Sample DAKOTA input for solving a least-squares problem.

An interesting feature of DAKOTA input, to which many users have become accustomed, is that the order of keywords given after a top-level keyword does not matter. For a simple example, both "cumulative distribution" and "distribution cumulative" have the same effect; the former seems more natural, but the latter would be required if input were required to be well ordered, i.e., if a lower-level keyword could only follow the keyword that enabled its presence, with no keywords enabled by a still higher-level keyword intervening.

This paper is about NIDR ("New Input Deck Reader"), which was written to simplify maintenance of DAKOTA, provide better checking of input, and allow use of aliases in that input. This paper explains NIDR and how it works, but this paper is not meant to provide detailed guidance to DAKOTA maintainers, who should consult the current DAKOTA Developer's Manual [9]. The rest of the paper is organized as follows. Section 2 introduces NIDR and its grammar file. Section 3 tells how NIDR parses "well-ordered" input, while Section 4 explains how NIDR turns unordered input into well-ordered input. Section 5 discusses *aliases*, a feature of NIDR meant to simplify input preparation. Section 6 describes a facility for sharing features common to several keywords, Section 7 summarizes some extensions, and Section 8 offers concluding remarks and gives pointers to the NIDR source. Appendix A summarizes the NIDR grammar.

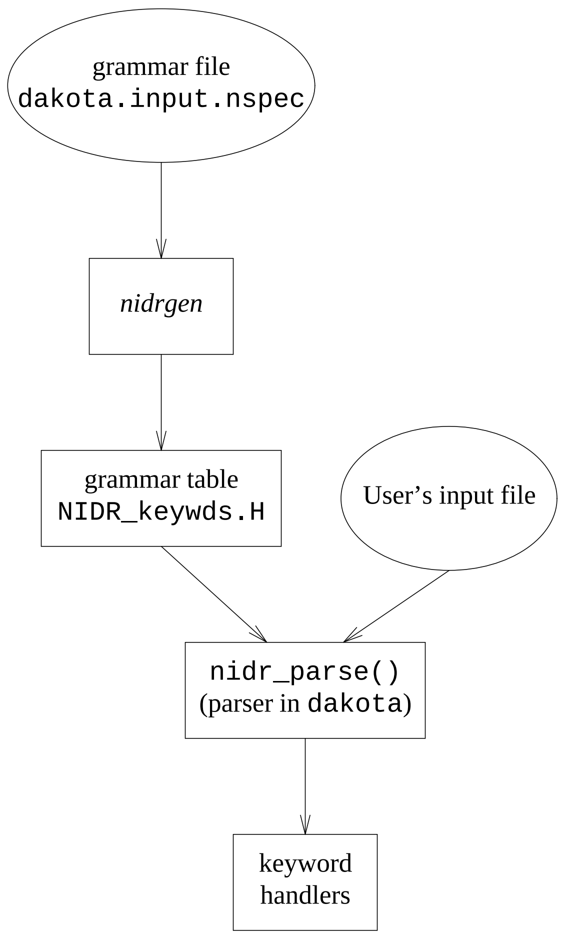

Figure 2: Overall operation of NIDR.

The overall operation of NIDR is illustrated in Figure 2, which mentions relevant DAKOTA source file names. The grammar file (`dakota.input.nspec`) describes the input that DAKOTA will accept. Program *nidrgen*, which is only run by DAKOTA developers in response to changes in the grammar file, turns the grammar file into a grammar table, `NIDR_keywds.H`, that tells DAKOTA's input parser, `nidr_parse()`, about the input it should accept. When DAKOTA starts execution, `nidr_parse()` reads the user's input and calls keyword handlers in response to the input, as specified in the grammar file. The keyword handlers suitably build and adjust DAKOTA data structures, in effect telling DAKOTA what to do.

2 IDR and NIDR

DAKOTA originally used a package called IDR [15] to specify and parse its input; IDR stands for "Input Deck Reader". Maintenance was cumbersome (see Chapter 11 of [7]), and most error checking required manual per-keyword coding. NIDR simplifies maintenance by reducing the manual effort needed to modify the input specification and by automating some error checking, e.g., ensuring that required keywords appear, that only one of a set of alternative keywords appears, and that values of the right sort (numeric or string) are given for keywords that need values, with no values appearing for keywords that do not require them.

Figure 3 shows a portion of the NIDR grammar for DAKOTA's top-level "responses" keyword. This portion specifies whether and how first derivatives are to be supplied: exactly one of the four keywords:

```
analytic_gradients
mixed_gradients
no_gradients
numerical_gradients
```

must be present. Some of these ("analytic_gradients" and "no_gradients") are simple keywords with no associated values or sub-keywords. The others have optional sub-keywords shown between square brackets ([and]), and keyword "mixed_gradients" also has two required sub-keywords:

```
id_analytic_gradients
id_numerical_gradients
```

In the grammar file, a keyword that enables other keywords and those other keywords are all placed between parentheses (if the keyword is required) or between square brackets (if optional). For instance, "mixed_gradients" and the keywords it enables are contained in parentheses, with "mixed_gradients" appearing immediately after the opening parenthesis.

Some of the keywords in Figure 3 have associated values, either a list of integers denoted by `INTEGERLIST` or a list of floating-point numbers denoted by `REALLIST`. Some keywords, such as the "convergence_tolerance" in Figure 1,

```
KEYWORD responses
        analytic_gradients
        |
        ( mixed_gradients
          id_analytic_gradients INTEGERLIST
          id_numerical_gradients INTEGERLIST
          [ fd_gradient_step_size REALLIST ]
          [ interval_type
            central
            | forward
            ]
          [ method_source
            dakota
            | vendor
            ]
          )
        | no_gradients
        |
        ( numerical_gradients
          [ fd_gradient_step_size REALLIST ]
          [ interval_type
            central
            | forward
            ]
          [ method_source
            dakota
            | vendor
            ]
          )
```

Figure 3: Grammar summary (partial) for `responses`.

only accept a single value, which in this case would be indicated by

`convergence_tolerance REAL`

in the grammar file. The grammar for "analysis_driver" (which also appears in Figure 1) would be "`analysis_driver STRING`". Some other keywords accept a list of strings, denoted by `STRINGLIST`.

DAKOTA ignores case in its input keywords. For simplicity, NIDR grammar files use lower case to introduce keywords and use reserved upper-case meta-keywords to indicate required values and introduce aliases (see §5 below).

Figure 3 shows DAKOTA input grammar in a form that is meant to be easy for users to understand. To control what actually happens during parsing, NIDR grammar files specify handlers, which are routines to be called when processing for a keyword begins (the "initial handler") and, for a keyword that introduces other keywords, when processing of contained keywords is finished (the "final handler", which is needed only if there are things to do that cannot be done until all contained keywords have been seen). An argument for each handler is usually also provided. As an example, Figure 4 shows a portion, corresponding to Figure 3, of the actual NIDR grammar file for DAKOTA (file `dakota.input.nspec` — see §8).

```
KEYWORD responses {N_rem3(start,0,stop)}
        analytic_gradients {N_rem(lit,gradientType_analytic)}
        |
        ( mixed_gradients {N_rem(lit,gradientType_mixed)}
          id_analytic_gradients INTEGERLIST {N_rem(intL,idAnalyticGrads)}
          id_numerical_gradients INTEGERLIST {N_rem(intL,idNumericalGrads)}
          [ fd_gradient_step_size REALLIST {N_rem(RealL,fdGradStepSize)} ]
          [ interval_type {0}
            central {N_rem(lit,intervalType_central)}
            | forward {N_rem(lit,intervalType_forward)}
            ]
          [ method_source {0}
            dakota {N_rem(lit,methodSource_dakota)}
            | vendor {N_rem(lit,methodSource_vendor)}
            ]
          )
        | no_gradients {N_rem(lit,gradientType_none)}
        |
        ( numerical_gradients {N_rem(lit,gradientType_numerical)}
          [ fd_gradient_step_size REALLIST {N_rem(RealL,fdGradStepSize)} ]
          [ interval_type {0}
            central {N_rem(lit,intervalType_central)}
            | forward {N_rem(lit,intervalType_forward)}
            ]
          [ method_source {0}
            dakota {N_rem(lit,methodSource_dakota)}
            | vendor {N_rem(lit,methodSource_vendor)}
            ]
          )
```

Figure 4: Actual grammar (partial) for `responses`.

Handlers and their arguments appear in braces. Up to four items may appear between the braces: the initial handler, its argument, the final handler, and its argument, with missing items taken to be zero and, for a handler, with zero meaning "no handler". Most of the entities between braces in Figure 4 are macro calls, which make the grammar file easier to read and supply abstruse C++ syntax that would be easy to get wrong if entered manually. A few keywords, such as "interval_type" and "method_source", serve only to enable the appearance of contained keywords and have no associated function calls, which is indicated by "0". The top-level "responses" keyword has both initial and final handlers, "start" and "stop", which the DAKOTA-specific macro `N_rem3` turns into `NIDRProblemDescDB::resp_start` and `NIDRProblemDescDB::resp_stop`; the former is called when processing of the responses keyword begins, and in this case it allocates a data structure that the sub-keywords manipulate.

It is instructive to examine Figure 5, which shows the implementation of the initial handler for DAKOTA's "responses" keyword. All the keyword handlers in NIDR client programs have the same signature: a pointer `keyname` to the keyword name, a pointer `val` to any associated value or values, a pointer `g` to a void pointer that the handler can set if desired, and another void pointer `v` for conveying details particular to this keyword.

```
 void NIDRProblemDescDB::
resp_start(const char *keyname, Values *val, void **g, void *v)
{
        if (!(*g = (void*)(new DataResponses)))
                botch("new failure in resp_start");
        }
```

Figure 5: Initial handler for DAKOTA's `responses` keyword.

In the case of Figure 5, a new `DataResponses` object is allocated and assigned to `*g`. The `g` argument passed to handlers for sub-keywords points to the `DataResponses` object thus allocated, so `*g` can act like a C++ "this" pointer to provide access to the surrounding context. The `v` argument to each handler is specified in the grammar file. Often it is a pointer to a member element that lets the handler adjust the appropriate field in the `DataResponses` object. For example, Figure 6 shows source for the keyword handler for `N_rem(lit,...)`, which appears twelve times in Figure 4. The final handler for "responses" (not shown) does some error checking and other postprocessing and makes the `DataResponses` object available to DAKOTA.

```
 struct
Resp_mp_lit {
        String DataResponses::* sp;
        const char *lit;
        };


 void NIDRProblemDescDB::
resp_lit(const char *keyname, Values *val, void **g, void *v)
{
        (*(DataResponses**)g)->*((Resp_mp_lit*)v)->sp
                = ((Resp_mp_lit*)v)->lit;
        }
```

Figure 6: Struct `Resp_mp_lit` and a handler that uses it.

Figure 6 begins with the declaration of a `struct` used in the handler named `resp_lit`. The first component of the `struct` is a pointer to a `String` member of `DataResponses`, and the second component provides the value that the handler assigns to this member. The `v` arguments to the `resp_lit` handler are pointers to `Resp_mp_lit` values. Here is how some of these values are declared:

```
#define MP2(x,y) resp_mp_##x##_##y = {&DataResponses::x,#y}
static Resp_mp_lit
        MP2(gradientType,analytic),
        MP2(gradientType,mixed),
        MP2(gradientType,none),
        MP2(gradientType,numerical),
        // ...
```

When a new `String`-valued member is added to `DataResponses`, one must simply add a corresponding line to the grammar file and to the above list of `Resp_mp_lit` values.

The `N_rem` macro used many times in Figure 4 is given by

```
#define N_rem(x,y) NIDRProblemDescDB::resp_##x,&resp_mp_##y
```

For example, `MP2(gradientType,analytic)` expands to

```
resp_mp_gradientType_analytic = {&DataResponses::gradientType,"analytic"}
```

and `N_rem(lit,gradientType_analytic)` expands to

```
NIDRProblemDescDB::resp_lit,&resp_mp_analytic
```

3 Table-driven Parsing

While inspired by the well known Unix tools *lex* and *yacc*, which let one specify code to be executed in response to indicated input, the way of specifying routines to be called with NIDR is sufficiently restricted that a simple table-driven parser can process input in client programs (like DAKOTA). (*Lex* and *yacc* are described many places; see, e.g., [11, 12, 16] for some pointers.) A parser-generator, *nidrgen* (itself a *lex* program), turns an NIDR grammar file into data structures for the NIDR parser in client programs. Figure 7, for instance, shows a few of the (nearly 1200) lines that *nidrgen* produces for DAKOTA.

Some `KeyWord` components enable the NIDR parser to provide error messages if more than one keyword in a set of alternatives appears, or if a required keyword fails to appear.

Parsing well-ordered input is straightforward. (Section 4 discusses an algorithm for well-ordering the input.) To get things started, *nidrgen* emits a `KeyWord` declaration with a fixed name for the top-level keywords, such as

```
KeyWord Dakota_Keyword_Top = "KeywordTop",0,6,0,0,Dakota::kw_186;
```

```
static KeyWord
//...
    kw_154[15] = {
            {"analytic_gradients",0,0,1,1,0,N_rem(lit,gradientType_analytic)},
            {"analytic_hessians",0,0,2,2,0,N_rem(lit,hessianType_analytic)},
            {"descriptors",7,0,5,0,0,N_rem(strL,responseLabels)},
            {"id_responses",3,0,4,0,0,N_rem(str,idResponses)},
            {"mixed_gradients",0,5,1,1,kw_137,N_rem(lit,gradientType_mixed)},
            // ...
            },
//...
    kw_186[6] = {
            {"interface",0,10,1,1,kw_8,N_ifm3(start,0,stop)},
            {"method",0,52,2,2,kw_111,N_mdm3(start,0,stop)},
            {"model",0,6,3,3,kw_134,N_mom3(start,0,stop)},
            {"responses",0,15,4,4,kw_154,N_rem3(start,0,stop)},
            {"strategy",0,9,5,5,kw_165,NIDRProblemDescDB::strategy_start},
            {"variables",0,19,6,6,kw_185,N_vam3(start,0,stop)}
            };
```

Figure 7: Sample of *nidrgen* output used in DAKOTA.

The basic NIDR parser maintains a stack of open keywords, which initially contains just an entry for the top-level keywords. When it sees a keyword that itself contains sub-keywords (such as "responses" or "mixed_gradients" in Figure 7), it adds an entry for the keyword to the top of the stack and, if present, calls the keyword's initial handler. When a keyword comes along that is not found among the sub-keywords of the keyword on top of the stack but is found lower down, the final handlers (if present) of top-of-stack keywords are called and the stack popped until the new keyword appears among the sub-keywords of the keyword at the stack top. *Nidrgen* has sorted each keyword's array of sub-keywords to permit binary searching of the sub-keywords (with inexact matching, as discussed below). The initial handler is called after any associated numeric or string values have been read, and such values are passed to the initial handler in its `val` argument.

4 Algorithm to Reorder Inputs

The NIDR parser, like its IDR predecessor, requires all keywords that pertain to a top-level keyword to follow that keyword and precede the next top-level keyword. (The original IDR required input for each top-level keyword to appear on one logical line, which meant one had to put backslashes at the end of physical lines when breaking a long logical line into several more manageable physical lines. For DAKOTA, we removed the requirement to use backslashes by treating the top-level keywords as reserved.)

To remove any need for special ordering of the lower-level keywords connected with a top-level keyword, the NIDR parser proceeds as follows. It maintains an AVL tree [1] of keywords that are reachable so far and of keywords that have been seen but are not yet reachable. (At first the NIDR parser used a hash table for this purpose, but DAKOTA allows one to abbreviate keywords. Use of an AVL tree permits inexact matching with searches that run in time proportional to the logarithm of the number of entries in the tree.) When it sees a reachable keyword, the NIDR parser attaches it to a list of keywords to be processed once processing of the containing keyword begins. If a newly found reachable keyword has associated sub-keywords, the sub-keywords are added to the AVL tree, and any hitherto unreachable keywords that match the new sub-keywords are added to the lists of keywords to be processed with those sub-keywords. In addition to appearing in the AVL tree, unreachable keywords are kept in a doubly-linked list of unreachable keywords, so when a keyword becomes reachable, it can be removed from the list of unreachable keywords in O(1) time. At the end of input, or when the next top-level keyword comes along, any keywords still in the list of unreachable keywords are reported as unknown.

5 Aliases

DAKOTA recognizes many classes of "variables", including

```
continuous_design
continuous_state
discrete_design
discrete_state
```

and many kinds of "uncertain" variables. Most have associated "descriptors" (string values) and bounds and often other quantities. With IDR, each such entity had to have its own unique name, e.g.,

```
cdv_descriptors
cdv_lower_bounds
cdv_upper_bounds
csv_descriptors
csv_lower_bounds
csv_upper_bounds
```

for "continuous_design" and "continuous_state" variables. NIDR allows each keyword to have one or more "aliases", which are alternate names that are treated the same as the original keyword. For instance, lines of the form

```
[ continuous_design INTEGER {...}
  [ cdv_descriptors ALIAS descriptors STRINGLIST {...} ]
  [ cdv_lower_bounds ALIAS lower_bounds REALLIST {...} ]
  [ cdv_upper_bounds ALIAS upper_bounds REALLIST {...} ]
  ]
[ continuous_state INTEGER {...}
  [ csv_descriptors ALIAS descriptors STRINGLIST {...} ]
  [ csv_lower_bounds ALIAS lower_bounds REALLIST {...} ]
  [ csv_upper_bounds ALIAS upper_bounds REALLIST {...} ]
  ]
```

in the grammar file allow use of the same aliases (where appropriate) for all the various classes of variables, which should allow simpler preparation of DAKOTA input files. For example, Figure 8 shows a portion of DAKOTA input [13] that was prepared before aliases could be used, followed by the same portion rewritten to use aliases.

6 Nesting with Multiple Left Parentheses

Many DAKOTA "methods" (i.e., solvers) share keywords. Originally with NIDR (and IDR) it was necessary to repeatedly specify such keywords. By generalizing the NIDR grammar to permit multiple left parentheses to be adjacent, we permit "factoring out" some common keywords. The general idea is to permit changing

```
(a b e f)
|
(c d e f)
```

into

```
((a b) | (c d) e f)
```

For example, Figure 9 shows a small portion of the grammar for DAKOTA with a common keyword, `actual_model_pointer`, factored out.

Among other things, *nidrgen* can pretty-print its input; the text in Figure 9 was pretty-printed this way, with an option to remove the {...} notation for function calls. As a debugging aid, the pretty printing can also expand factored keywords to a form with no adjacent left parenthesis. This is useful when manually changing the grammar file to factor common elements out; the corresponding expanded, pretty-printed output should not change.

7 Extensions for DAKOTA 5.0 and Jaguar

In preparing the 5.0 release of DAKOTA [3], we found it convenient to extend NIDR in several ways, such as allowing the grammar specification to give

Without aliases:
```
        normal_uncertain = 2
          nuv_means          =  30. 500000.
          nuv_std_deviations  =  10.  50000.
          nuv_descriptor      = 'F0'  'P1'
        lognormal_uncertain = 4
          lnuv_means          = 300.  20. 300. 400.
          lnuv_std_deviations =   3.   2.   5.  35.
          lnuv_descriptor     =  'B'  'D'  'H' 'Fs'
        gumbel_uncertain = 2
          guuv_alphas         = 1.4250554e-5 1.4250554e-5
          guuv_betas          = 559496.31    559496.31
          guuv_descriptor     =  'P2'    'P3'
        weibull_uncertain = 1
          wuv_alphas          =     5.7974
          wuv_betas           = 22679.4777
          wuv_descriptor      =    'E'
```
With aliases:
```
        normal_uncertain = 2
          means           =  30. 500000.
          std_deviations  =  10.  50000.
          descriptor      = 'F0'  'P1'
        lognormal_uncertain = 4
          means          = 300.  20. 300. 400.
          std_deviations =   3.   2.   5.  35.
          descriptor     =  'B'  'D'  'H' 'Fs'
        gumbel_uncertain = 2
          alphas         = 1.4250554e-5 1.4250554e-5
          betas          = 559496.31    559496.31
          descriptor     =  'P2'    'P3'
        weibull_uncertain = 1
          alphas          =     5.7974
          betas           = 22679.4777
          descriptor      =    'E'
```

Figure 8: DAKOTA input [13] without and with aliases.

bounds on numeric values and allowing default values to be given, with the option of defining preprocessor symbols to associate names with default values, so the names can appear in DAKOTA source code instead of hard-coded numeric values. Thus default values can be changed just by editing the grammar file, automatically regnerating derived files, and recompiling.

Associated with the 5.0 release of DAKOTA is a graphical user interface called *Jaguar* [10]. Jaguar 1.0 appeared with DAKOTA 4.0 and was hard to maintain. A major rewrite of Jaguar was associated with DAKOTA 5.0, giving Jaguar 2.0, which is based on Eclipse [6]. For ease of maintenance, we extended NIDR to generate a grammar description for Jaguar that is augmented with display hints, URLs documenting various keywords, and short descriptions. This extra information is not used by DAKOTA itself, and we found it convenient to convey the extra information to *nidrgen* in a separate file (called `$DAKOTA/src/dakota.input.desc` in the DAKOTA source). Brian

```
( ( local
    taylor_series
    )
  |
  ( multipoint
    tana
    )
  actual_model_pointer STRING
  )
```

Figure 9: Some factoring of keywords.

Adams wrote a script to extract material for the "`.desc`" file from DAKOTA documentation sources. Items in this file are connected with entries in the grammar specification file by "TAG" names. TAG names can appear in both files, but we found it convenient to assume default TAG values in the grammar specification file, obtained by concatenating keyword names and slashes, as in "`interface/analysis_drivers/fork`".

Conventionally, several new keywords and associated values can appear in the "`.desc`" file. Jaguar displays all keywords together that have the same GROUP string value. A DESC string provides a short description and, optionally, a URL for more details. A LEN specification gives the name of a keyword whose associated value is the number of items that should appear in a list (e.g., of descriptors, bounds, or starting values); variant LEN1 means a single value is an acceptable alternative that causes all items in a list to have that value.

Jaguar reads and writes conventional DAKOTA input files. When reading, it requires well-ordered input. We used NIDR facilities to make a program, `dakreorder`, that well-orders an input file. One issue for maintenance is that `dakreorder` has a DAKOTA grammar specification compiled into it, which means it needs to be recompiled when the input grammar is extended or modified. This led us to create a variant of `dakreorder` called `dakreord` that reads a grammar specification summary, which *nidrgen* can now write. Now when Jaguar starts up, it looks to see if there is an updated grammar specification file — one newer than the current specification summary file. If so, it transparently runs *nidrgen* to obtain an updated specification summary. Thus as DAKOTA development continues, Jaguar automatically adapts.

8 Conclusion

NIDR appeared with Version 4.1 of DAKOTA. Preliminary experience in this context is favorable. The NIDR parser is smaller than its IDR predecessor, but provides more error checking and simplifies maintenance. For DAKOTA users, the chance to use aliases should simplify input preparation. NIDR could easily be used by other programs.

In the DAKOTA source (see [2]), source for *nidrgen* appears in directory `Dakota/packages/nidr`. The other files relevant to NIDR appear in directory `Dakota/src`: the grammar file is called `dakota.input.nspec`; *nidrgen* reads this grammar file and writes file `NIDR_keywds.H`; source for the NIDR parser is files `nidr.c` and `nidr.h`. Dakota-specific code appears in `NIDRProblemDescDB.H` and `NIDRProblemDescDB.C`. The latter `#include`s `NIDR_keywds.H` and invokes `nidr_parse(parser)`, in which `parser` is a character string possibly given by command-line option `--parser` to DAKOTA: the default is "`nidr`"; specifying "`nidrstrict`" turns off the well-ordering algorithm, and "`nidr:`*filename*" causes the input to be written to *filename* after it has been well ordered. Extensions to NIDR occasioned by work on the DAKOTA 5.0 release and an associated major revision of the Jaguar GUI for DAKOTA now make it possible for Jaguar to adapt automatically to ongoing DAKOTA development.

Acknowledgment. I thank Brian Adams and Laura Swiler for helpful comments on the manuscript.

References

[1] `http://www.nist.gov/dads/HTML/avltree.html`.

[2] `http://www.cs.sandia.gov/DAKOTA/`.

[3] `http://www.cs.sandia.gov/DAKOTA/release_notes.html`.

[4] J. E. Dennis, D. M. Gay, and R. E. Welsch. ALGORITHM 573: NL2SOL–an adaptive nonlinear least-squares algorithm. *ACM Trans. Math. Software*, 7:369–383, 1981.

[5] DAKOTA uses PORT versions of NL2SOL; see `http://www.netlib.org/port/readme` or `http://netlib.sandia.gov/port/readme.gz`.

[6] `http://www.eclipse.org/`.

[7] Michael S. Eldred et al. DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 4.0 developers manual. Report SAND2006-0456, Sandia National Laboratories, 2006. `http://www.cs.sandia.gov/DAKOTA/licensing/release/Developers4.0.pdf`.

[8] Michael S. Eldred et al. DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 4.1 developers manual. Report SAND2006-6337, Sandia National Laboratories, 2007. `http://www.cs.sandia.gov/DAKOTA/licensing/release/Users4.1.pdf`.

[9] Michael S. Eldred et al. DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 4.1+ developers manual.

Version of the day (votd), Sandia National Laboratories, 2008. http://www.cs.sandia.gov/DAKOTA/licensing/votd/html-dev/index.html.

[10] http://www.cs.sandia.gov/dakota/licensing/download_jaguar_2_0.html.

[11] http://en.wikipedia.org/wiki/Lex_programming_tool.

[12] The lex and yacc page. http://dinosaur.compilertools.net/.

[13] File Dakota/test/dakota_rbdo_steel_column.in in source distribution file Dakota_4_1.src.tar.gz, available from http://www.cs.sandia.gov/DAKOTA/.

[14] H. H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *Computer J.*, 3:175–184, 1960.

[15] Joe R. Weatherby, James A. Schutt, James S. Peery, and Roy E. Hogan. Delta: An object-oriented finite element code architecture for massively parallel computers. Report SAND96-0473, Sandia National Laboratories, 1996.

[16] http://en.wikipedia.org/wiki/Yacc.

9 Appendix A: Summary of NIDR Grammar

It may be helpful to see a summary of the grammar accepted by *nidrgen*, the NIDR parse-table generator. An informal summary appears in Figures 10 and 11, with grammatical elements in *italics*, optional elements subscripted by *opt*, and alternatives appearing on separate lines. One possibility for a *keywordname*, for instance, is $identifier$ ALIAS $identifier$. An *identifier* is a sequence of lower-case letters, digits, and underscores that starts with a letter. An *hstring* is a sequence of most printable characters other than braces and describes a keyword handler, its argument, or sequence thereof, and $hstring_{1-4}$ means one to four *hstring*s appearing on successive lines. Reserved words are in UPPER-CASE. Special characters appear between single quotes (such as '(' and ')' for left and right parentheses.) The last grammatical element, *nidrgen-input*, is what *nidrgen* reads.

keywordname :

identifier

keywordname ALIAS *identifier*

type :

INTEGER

REAL

STRING

INTEGERLIST

REALLIST

STRINGLIST

action :

'{' $hstring_{1-4}$ '}'

tag :

TAG *quoted_string*

group :

GROUP *quoted_string*

desc :

DESC *quoted_string*

len :

LEN *identifier*

LEN1 *identifier*

default_val :

$:= defname_opt$ *number*

$:= defname_opt$ *quoted_string*

bound :

$\geq$ *number*

$>$ *number*

$<$ *number*

$\leq$ *number*

optional_item :

type

action

tag

group

desc

len

default_val

bound

Figure 10: Summary of *nidrgen* input grammar (part 1).

optional_items :

optional_item
optional_items optional_item

simplekeyword :

keywordname $optional\_items_opt$

initialkeyword :

simplekeyword
reqkeywordlist
initialkeyword '|' *simplekeyword*
initialkeyword '|' *reqkeywordlist*

reqkeywordlist :

'(' *initialkeyword* $keywordlist_{opt}$ ')'

optkeywordlist :

'[' *initialkeyword* $keywordlist_{opt}$ ']'

keyword :

initialkeyword
reqkeywordlist
optkeywordlist

keywordlist :

keyword
keywordlist keyword

toplevel-keyword:

KEYWORD *simplekeyword* $keywordlist_{opt}$

nidrgen-input:

toplevel-keyword
nidrgen-input ';'
nidrgen-input toplevel-keyword

Figure 11: Summary of *nidrgen* input grammar (part 2).